Exhibit C

In-App Purchase Programming Guide

Developer

Contents

Introduction 5
At a Glance 5
Understand Products and Design Your App 5
All Apps Implement Core Behavior 6
Some Apps Implement Additional Behavior 6
Apps and Products are Submitted for Review 6
See Also 6

Designing Your App's Store 7
Understanding What You Can Sell Using In-App Purchase 7
Creating Products in iTunes Connect 8
Types of Product 8
Differences Between Product Types 9

Displaying Your App's Store UI 11
Getting a List of Product IDs 11
Embedding Product IDs in the App Bundle 12
Fetching Product IDs from Your Server 12
Retrieving Product Information 14
Present Your Store UI 15

Requesting Payment 17
Creating a Payment Request 17
Submitting a Payment Request 18
Suggested Testing Steps 18
Sign In to the iTunes Store with Your Test Account 19
Verify Your Observer Code 19
Test Fetching the List of Products 19
Test a Products Request 20
Test a Payment Request 20

Delivering Products 21
Waiting for the Store to Process Transactions 21
Persisting the Purchase 23

Persisting Using User Defaults or iCloud 23
Persisting Using Your Own Server 25
Making the Product Available 25
Finishing the Transaction 25
Suggested Testing Steps 26
Test a Successful Transaction 26
Verify That Transactions Are Finished 26

Providing Purchased Content 27
Unlocking Local Content 28
Downloading Hosted Content from Apple's Server 28
Downloading Content from Your Own Server 29
Locating and Managing Downloaded Content 29

Working with Subscriptions 30
Calculating a Subscription's Active Period 30
Expiration and Renewal 31
Cancellation 32
Cross-Platform Considerations 32
The Test Environment 33

Preparing for App Review 34
Submitting Products for Review 34
Receipts in the Test Environment 34

Document Revision History 36

Figures, Tables, and Listings

Designing Your App's Store 7
Table 1-1 Comparison of product types 9

Displaying Your App's Store UI 11
Figure 2-1 Stages of the purchase process—displaying store UI 11
Table 2-1 Comparison of approaches for obtaining product identifiers 11
Listing 2-1 Retrieving product information 14
Listing 2-2 Formatting a product's price 16

Requesting Payment 17
Figure 3-1 Stages of the purchase process—requesting payment 17
Listing 3-1 Creating a payment request 17

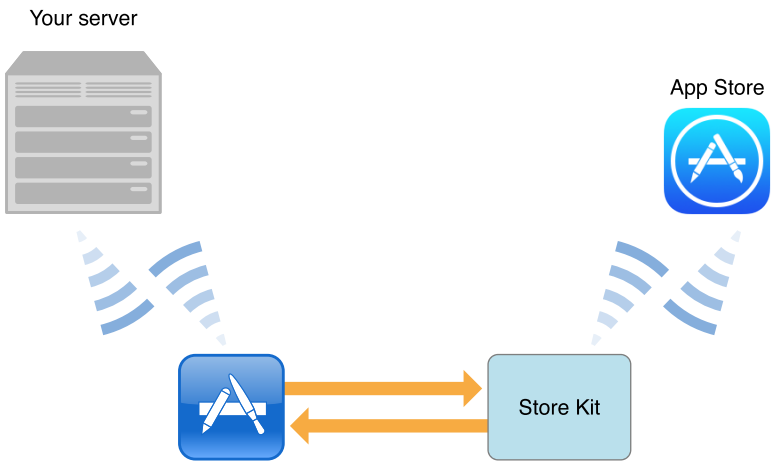
Delivering Products 21
Figure 4-1 Stages of the purchase process—delivering products 21
Table 4-1 Transaction statuses and corresponding actions 21
Listing 4-1 Responding to transaction statuses 22

Working with Subscriptions 30
Figure 6-1 Example subscription timeline 30
Table 6-1 Timeline of a sample subscription 30

Preparing for App Review 34
Figure 7-1 Development, review, and production environments 34

Introduction

In-App Purchase allows you to embed a store inside your app using the Store Kit framework. This framework connects to the App Store on your app's behalf to securely process payments from users, prompting them to authorize payment. The framework then notifies your app, which provides the purchased items to users. Use In-App Purchase to collect payment for additional features and content.



At a Glance

For example, using In-App Purchase, you can implement the following scenarios:

- A basic version of your app with additional premium features
- A magazine app that lets users purchase and download new issues
- A game that offers new environments (levels) to explore
- An online game that allows players to purchase virtual property

Understand Products and Design Your App

Understanding what kinds of products and behaviors are supported by In-App Purchase lets you design your app and in-app store to make the best use of this technology.

Relevant Sections: "Designing Your App's Store" (page 7)

All Apps Implement Core Behavior

All products require certain basic behavior in your app to support the basic In-App Purchase flow of letting the user select a product, requesting payment for the product, and delivering the purchased product.

Relevant Sections: "Displaying Your App's Store UI" (page 11), "Requesting Payment" (page 17), "Delivering Products" (page 21)

Some Apps Implement Additional Behavior

Some products require aditional behavior app: products that include digital assets need your app to perform the download, and subscriptions need your app to keep track of when the user has an active subscription.

Relevant Sections: "Providing Purchased Content" (page 27), "Working with Subscriptions" (page 30)

Apps and Products are Submitted for Review

When you are done developing and testing, you submit your app and your In-App Purchase products for review.

Relevant Sections: "Preparing for App Review" (page 34)

See Also

- *iTunes Connect Developer Guide* describes how to use iTunes Connect, in particular, how to create and configure your app's products.
- *Receipt Validation Programming Guide* describes how to work with receceipts, in particular, the record of sucessful in-app purchases.

Designing Your App's Store

The first step in designing your app's store is to determine which types of product you need to use to build the user experience you designed for in your app. A **product** is something you want to sell in your app's store. You create and configure products in iTunes Connect, and your app interacts with products using the `SKProduct` and `SKProductsRequest` classes.

Understanding What You Can Sell Using In-App Purchase

You can use In-App Purchase to sell content, app functionality, and services:

- **Content.** Deliver digital content or assets, such as magazines, photos, and artwork. Content can also be used by the app itself—for example, additional characters and levels in a game, filters in a camera app, and stationery in a word processor.
- **App functionality.** Unlock behavior, and expand features you've already delivered. For example, a free game that offers multiplayer mode as an in-app purchase and a free weather app that lets users make a one-time purchase to remove ads.
- **Services.** Have users pay for one-time services such as voice transcription, and for ongoing services such as access to a collection of data.

You *can't* use In-App Purchase to sell real-world goods and services or unsuitable content:

- **Real-world goods and services.** You must deliver a digital good or service within your application when using In-App Purchase. Use a different payment mechanism to let your users buy real-world goods and services in your app, such as a credit card or payment service.
- **Unsuitable content.** Do not use In-App Purchase to sell content that the App Review Guidelines do not allow, for example, pornography, hate speech, or defamation.

For detailed information about what you can offer using In-App Purchase, see your license agreement and the App Review Guidelines. Reviewing the guidelines carefully before you start coding helps you avoid delays and rejection during the review process. If the guidelines don't address your case in sufficient detail, you can ask the App Review team specific questions using the online contact form.

After you know what products you want to sell in your app and you determine that In-App Purchase is the appropriate way to sell those products, you need to create the products in iTunes Connect.

Creating Products in iTunes Connect

Before you can start testing code, you need to configure products for your app to interact with. You can configure products in iTunes Connect at any time. As you develop your app, expect to configure a few test products early on, and come back later to configure additional products as your app's design develops.

Products are reviewed when you submit your app as part of the app review process. Before users can buy a product, it must be approved by the reviewer and you must mark it as "cleared for sale" in iTunes Connect.

A typical timeline for a new app might look as follows:

1. Create products in iTunes Connect.
2. Write code that uses those products in test environment.
3. Refine your app design, and go back into iTunes Connect to create additional products or adjust the existing products.
4. Continue iterating, refining, and testing, both your app's code and the products.
5. Submit the app and products for review.

For details about configuring products in iTunes Connect, see "In-App Purchase" in *iTunes Connect Developer Guide*.

Types of Product

Product types let you use In-App Purchase in several different situations, by letting the products behave in different ways that are appropriate for specific use cases. In iTunes Connect, you select one of the following product types:

- **Consumable products.** Items that get used up over the course of running your app. Examples include minutes for a Voice over IP app and one-time services such as voice transcription.
- **Non-consumable products.** Items that remain available to the user indefinitely on all of the user's devices. They are made available to all of the user's devices. Examples include content, such as books and game levels, and additional app functionality.
- **Auto-renewable subscriptions.** Episodic content. Like non-consumable products, auto-renewable subscriptions remain available to the user indefinitely on all of the user's devices. Unlike non-consumable products, auto-renewable subscriptions have an expiration date. You deliver new content regularly, and users get access to content published during the time period that their subscription is active. When an auto-renewable subscription is about to expire, the system automatically renews it on the user's behalf.

- **Non-renewable subscriptions.** Subscriptions that don't involve delivering episodic content. Examples include access to a database of historic photos or a collection of flight maps. It is your app's responsibility to make the subscription available on all of the user's devices and to let users restore the purchase. This product type is often used when your users already have an account on your server, which you can use to identify them when restoring content. Expiration and the duration of the subscription are also left to your app (or your server) to implement and enforce.
- **Free subscriptions.** A way to put free subscription content in Newsstand. After a user signs up for a free subscription, the content is available on all devices associated with the user's Apple ID. Free subscriptions do not expire and can be offered only in Newsstand-enabled apps.

Differences Between Product Types

Each product type is designed for a particular use case. The behavior of different products differs in certain ways, as shown in Table 1-1 (page 9). For example, consumable products get used up and subscriptions expire, so the store lets the user buy the products of these types multiple times.

Table 1-1 Comparison of product types

Product type	Consumable	Non-consumable	Auto-renewable	Non-renewing	Free Subscription
Users can buy	Multiple times	Once	Multiple times	Multiple times	Once
Synced across devices	Not synced	By the system	By the system	By your app	By the system
Restored	Not restored	By the system	By the system	By your app	By the system
Appears in the receipt	Once	Always	Always	Once	Always

Products that expire or get used up—consumable products, auto-renewable subscriptions, and non-renewing subscriptions—can be purchased multiple times to get the consumable item again or extend the subscription. Non-consumable products and free subscriptions unlock content that remains available to the user indefinitely, so these can only be purchased once.

Products that do not get used up are made available across all of the user's devices. These products are also restored so users can continue to access their purchased content even after erasing a device or buying a new device. Consumable products by their nature are not meant to be synced or restored. Users understand that, for example, buying ten more bubbles on their iPhone does not also give them ten bubbles on their iPad.

Store Kit handles the syncing and restoring process for auto-renewable and free subscriptions, and for non-consumable products. Your app is responsible for syncing and restoring non-renewable subscriptions.

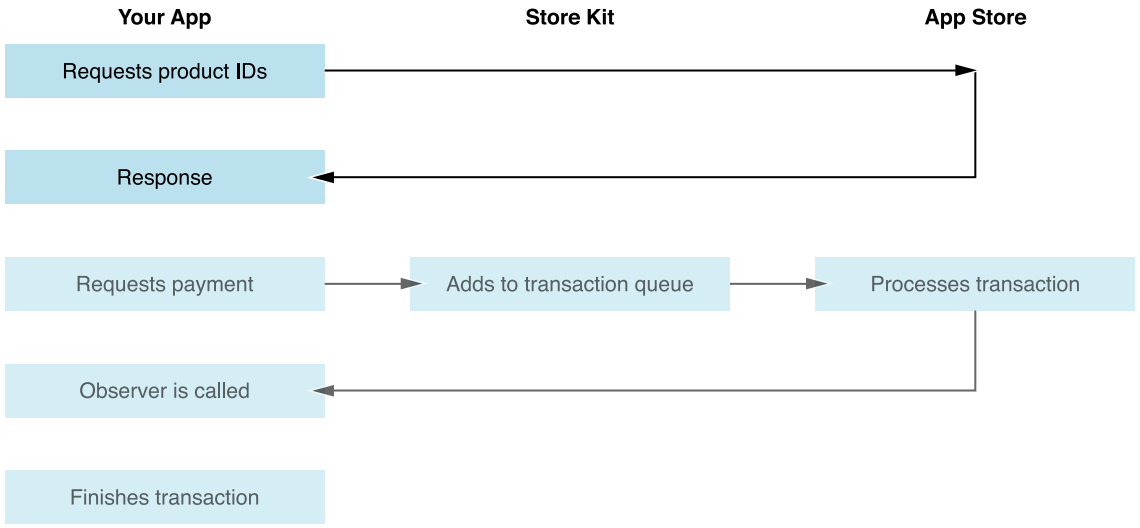
Non-renewing subscriptions differ from auto-renewable subscriptions in a few key ways which give your app the flexibility to implement the correct behavior for your needs:

- Your app is responsible for calculating the time period that the subscription is active, and determining what content needs to be made available to the user.
- Your app is responsible for detecting that a subscription is approaching its expiration date and prompting the user to renew the subscription by purchasing the product again.
- Your app is responsible for making subscriptions available across all the user's devices after they are purchased. For example, most subscriptions are provided by a server; your server would need some mechanism to identify users and associate subscription purchases with the user who purchased them.

Displaying Your App's Store UI

In the first part of the purchase process, your app presents its store UI to the user, and then lets the user selects a product, as shown in Figure 2-1. The second and third part of the process are described in the two chapters that follow this one.

Figure 2-1 Stages of the purchase process—displaying store UI



Getting a List of Product IDs

To get a list of product identifiers, have your app either read them from a file in your app bundle or fetch them from your server. Table 2-1 summarizes the differences between the two approaches. If you offer a fixed list of products, such as an in-app purchase to remove ads, embed the list in your app bundle. If your list of product identifiers can change without you updating your app, have your app fetch the list from your server.

Table 2-1 Comparison of approaches for obtaining product identifiers

	Embedded in the app bundle	Fetched from your server
Used for purchases that	Unlock functionality	Deliver content
List of products can change	When the app is updated	At any time
Requires a server	No	Yes

There is no runtime mechanism to fetch a list of all products configured for an app in iTunes Connect. You are responsible for managing your app's list of products and providing that information to your app.

Embedding Product IDs in the App Bundle

Include a property list file in your app bundle containing an array of product identifiers, such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>com.example.level1</string>
    <string>com.example.level2</string>
    <string>com.example.rocket_car</string>
</array>
</plist>
```

To get product identifiers from the property list, locate the file in the app bundle and read it.

```
NSURL * url = [[NSBundle mainBundle] URLForResource:@"product_ids"
                                      withExtension:@"plist"];
NSArray * productIdentifiers = [NSArray arrayWithContentsOfURL:url];
```

Fetching Product IDs from Your Server

Store a JSON file on your server with the product identifiers. For example:

```
[
    "com.example.level1",
    "com.example.level2",
    "com.example.rocket_car"
]
```

To get product identifiers from your server, fetch the JSON file and read it.

```
-fetchProductIdentifiersFromURL:(NSURL*)url delegate(id):delegate
{
    dispatch_queue_t global_queue =
           dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0);
    dispatch_async(global_queue, ^{
        NSError * err;
        NSData * jsonData = [NSData dataWithContentsOfURL:url
                                                  options:NULL
                                                    error:&err];
        if (!jsonData) {
           // Handle the error...
        }

        NSArray * productIdentifiers = [NSJSONSerialization
            JSONObjectWithData:jsonData options:NULL error:&err];
        if (!productIdentifiers) {
           // Handle the error...
        }

        dispatch_queue_t main_queue = dispatch_get_main_queue();
        dispatch_async(main_queue, ^{
           [delegate displayProducts:productIdentifiers];
        }
    });
}
```

For information about downloading files using `NSURLConnection`, see "Using NSURLConnection" in *URL Loading System Programming Guide*.

To ensure that your app remains responsive, use a background thread to download the JSON file and extract the list of product identifiers. To minimize the data transferred, use standard HTTP caching mechanisms, such as the `Last-Modified` and `If-Modified-Since` headers.

Retrieving Product Information

To make sure your users see only products that are actually available for purchase, query the App Store before displaying your store UI.

Use a products request object to query the App Store. First, create an instance of `SKProductsRequest` and initialize it with a list of product identifiers. The products request retrieves information about valid products, along with a list of the invalid product identifiers, and then calls its delegate to process the result. The delegate must implement the `SKProductsRequestDelegate` protocol to handle the response from the App Store. Listing 2-1 shows a simple implementation of both pieces of code.

Listing 2-1 Retrieving product information

```
// Custom method
- validateProductIdentifiers:(NSArray *)productIdentifiers
{
    SKProductsRequest productsRequest = [[SKProductsRequest alloc]
        initWithProductIdentifiers:[NSSet setWithArray:productIdentifiers]];
    productsRequest.delegate = self;
    [productsRequest start];
}

// SKProductsRequestDelegate protocol method
- (void)productsRequest:(SKProductsRequest *)request
     didReceiveResponse:(SKProductsResponse *)response
{
    self.products = response.products;

   for (NSString * invalidProductIdentifier in response.invalidProductIdentifiers)
 {
        // Handle any invalid product identifiers.
    }

    [self displayStoreUI]; // Custom method
}
```

When the user purchases a product, you need the corresponding product object to create a payment request, so keep a reference to the array of product objects that is returned to the delegate. If the list of products your app sells can change, you may want to create a custom class that encapsulates a reference to the product object as well as other information—for example, pictures or description text that you fetch from your server.

The way you handle invalid product identifiers depends on where the list of product identifiers came from. If your app loaded the list from its app bundle, log a note to aid debugging and ignore the invalid identifiers. If your app fetched the list from your server, let your server know which product identifiers were invalid. As with all other interactions between your app and your server, the details and mechanics of this are up to you.

Present Your Store UI

The design of your store has an import impact on your in-app sales, so it's worth investing the time and effort to get it right. Design the user interface for your app's store so that it integrates with the rest of your app. Store Kit can't provide a store UI for you. Only you know your app and its content well enough to design your store UI so it showcases your products in their best light and integrates with the rest of your app.

Consider the following guidelines as you design and implement your app's store UI:

Display a store only if the user can make payments. To determine whether the user can make payments, call the `canMakePayments` class method of the `SKPaymentQueue` class. If the user can't make payments (for example, because of parental restrictions), either tell the user that the store is not currently supported (by displaying appropriate UI) or omit the store portion of your UI entirely.

Expose as much information about a product as possible. Users want to know exactly what they're going to buy. If possible, let users interact with a product. For example, a game that lets the user buy new characters can let users run a short course as the new character. Likewise, a drawing app that lets the user buy additional brushes can let users draw with the new brush on a small scratch pad. This kind of design gives users an opportunity to experience the product and be convinced they want to buy it.

Use the properties of your product objects to populate your store's user interface. Their properties give you information such as the product's price and its description. Combine this with additional data from your server or the app bundle, such as images or demos of your products.

Display prices using the locale and currency returned by the store. Don't try to convert the price to a different currency in your UI, even if the user's locale and the price's locale are different. Consider, for example, an English-speaking user in China who pays for purchases with a Chinese credit card but prefers the US English locale settings for apps. You app would display its UI according to the user's locale, using US English as requested by the user. It would display prices according to the products' locale, using the currency that corresponds to

the Chinese locale (renminbi) with that locale's formatting. Converting the prices to US dollars (to match the US English locale) would be misleading because the user is billed in renminbi, not US dollars. Listing 2-2 shows how to correctly format a price using the product's locale information.

Listing 2-2 Formatting a product's price

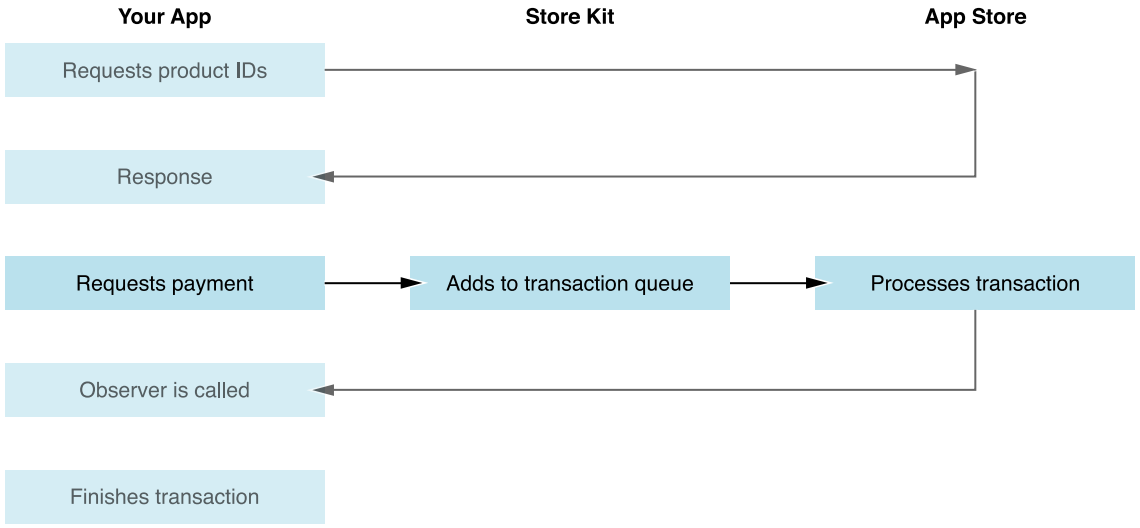
```
NSNumberFormatter * numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[numberFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
[numberFormatter setLocale:product.priceLocale];
NSString * formattedPrice = [numberFormatter stringFromNumber:product.price];
```

After a user selects a product to buy, your app connects to the store to request payment for the product.

Requesting Payment

In the second part of the purchase process, after the user has chosen to purchase a particular product, your app submits payment request to the App Store, as shown in Figure 3-1 (page 17).

Figure 3-1 Stages of the purchase process—requesting payment



Creating a Payment Request

When the user selects a product to buy, create a payment request using a product object. Populate the quantity if needed, and add the payment to the transaction queue to submit it to the App Store for processing. Adding a payment object to the queue multiple times results in multiple transactions—the user will be charged multiple times and your app will be expected to deliver the product multiple times. Listing 3-1 shows a simplified example of the process.

Listing 3-1 Creating a payment request

```
// At app launch
[[SKPaymentQueue defaultQueue] addTransactionObserver:observer];

// ...
```

```
// When the user buys a product
SKPayment payment = [SKPayment paymentWithProduct:product];
payment.quantity = 2;
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

Store Kit calls the transaction queue observer to handle a completed or failed payment request. For information about implementing the observer, see "Providing Purchased Content" (page 27).

Submitting a Payment Request

The transaction queue plays a central role in letting your app communicate with the App Store through the Store Kit framework. You add work to the queue that the App Store needs to act on, such as a payment request that needs to be processed. After the App Store has processed the request, Store Kit calls your transaction queue observer to handle the result. In addition to using the transaction queue for payment requests, your app also uses the transaction queue to download hosted content and to find out that subscriptions have been renewed, as discussed in "Providing Purchased Content" (page 27) and "Working with Subscriptions" (page 30).

Register your observer when your app is launched, and make sure that the observer is ready to handle a transaction at any time, not only after you add a transaction to the queue. For example, consider the case of a user buying something in your app right before going into a tunnel. Your app isn't able to deliver the purchased content because there is no network connection. The next time your app is launched, Store Kit calls your transaction queue observer again and delivers the purchased content at that time. Similarly, if your app fails to mark a transaction as finished, Store Kit calls the observer every time that your app is launched until the transaction in properly marked as finished.

Suggested Testing Steps

Test each part of your code to verify that you have implemented it correctly.

Prerequisite: To perform the suggested tests, you must have at least one product configured for your app in iTunes Connect. For more information, see "Creating Products in iTunes Connect" (page 8).

Sign In to the iTunes Store with Your Test Account

Create a test user account in iTunes Connect, as described in *iTunes Connect Developer Guide*.

On a development iOS device, sign out of the iTunes Store in Settings. Then build and run your app from Xcode.

On a development OS X device, sign out of the Mac App Store. Then build your app in Xcode and launch it from Finder.

Use your app to make an in-app purchase. When prompted to sign in to the App Store, use your test account. Note that the text "[Environment: Sandbox]" appears as part of the prompt, indicating that you are connected to the test environment.

If the text "[Environment: Sandbox]" does not appear, that indicates you are using the production environment. Make sure you are running a development signed build of your app. Production signed builds use the production environment.

Important: Do not use your test user account to sign in to the production environment; if you do, the test user account becomes invalid and can no longer be used.

Verify Your Observer Code

Review the transaction observer's implementation of the `SKPaymentTransactionObserver` protocol. Verify that it can handle transactions even if you aren't currently displaying your app's store UI and even if you didn't recently initiate a purchase.

Locate the call to the `addTransactionObserver:` method of `SKPaymentQueue` in your code. Verify that your app calls this method at app launch.

Test Fetching the List of Products

If your product identifiers are embedded in your app, set a breakpoint in your code after they are loaded and verify that the instance of `NSArray` contains the expected list of product identifiers.

If your product identifiers are fetched from a server, manually fetch the JSON file—using a web browser such as Safari or a command-line utility such as `curl`—and verify that the data returned from your server contains the expected list of product identifiers. Also verify that your server correctly implements standard HTTP caching mechanisms.

Test a Products Request

Using the list of product identifiers that you tested, create and submit an instance of `SKProductsRequest`. Set a breakpoint in your code, and inspect the lists of valid and invalid product identifiers. If there are invalid product identifiers, review your products in iTunes Connect and correct your JSON file or property list.

Test a Payment Request

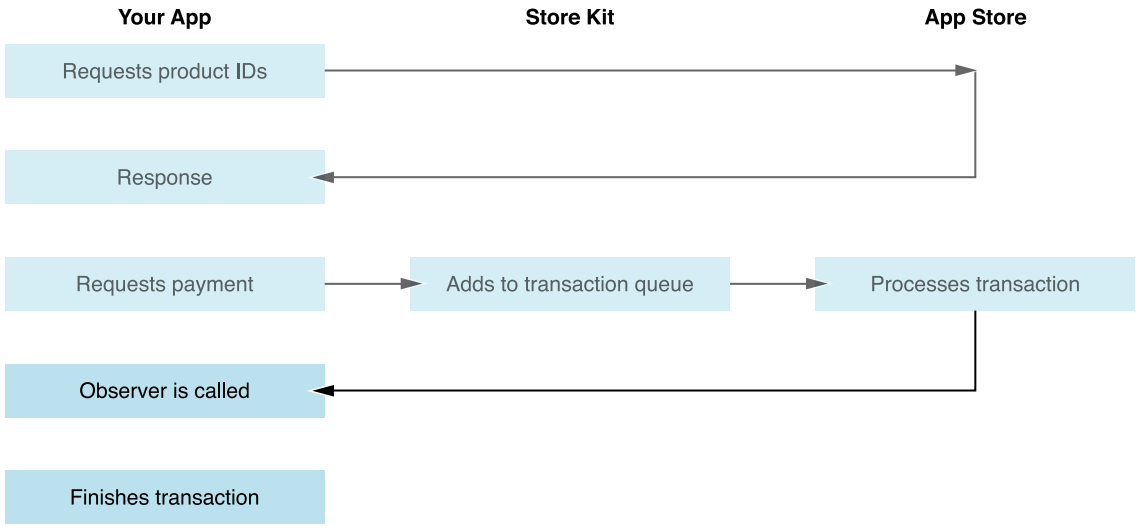
Create an instance of `SKPayment` using a valid product identifier that you have already tested. Set a breakpoint and inspect the payment request. Add the payment request to the transaction queue, and set a breakpoint to confirm that the `paymentQueue:updatedTransactions:` method of your observer is called.

During testing, it's ok to finish the transaction immediately without providing the content. However, even during testing, failing to finish the transaction can cause problems: unfinished transaction remain in the queue indefinitely, which could interfere with later testing.

Delivering Products

In the last part of the purchase process, after the App Store has processed the payment request, your app stores information about the purchase for future launches, downloads the purchased content, and marks the transaction as finished, as shown in Figure 4-1 (page 21).

Figure 4-1 Stages of the purchase process—delivering products



Waiting for the Store to Process Transactions

Implement the `paymentQueue:updatedTransactions:` method on your transaction queue observer. Store Kit calls this method when the status of a transaction changes—for example, when a payment request has been processed. The transaction status tells you what action your app needs to perform, as shown in Table 4-1 and Listing 4-1.

Table 4-1 Transaction statuses and corresponding actions

Status	Meaning	Action to take in your app
`SKPaymentTransaction-StatePurchasing`	Transaction still in progress	Update your UI to reflect the status, and wait to be called again.
`SKPaymentTransaction-StateFailed`	Transaction failed	Use the value of the `error` property to present a message to the user.

Status	Meaning	Action to take in your app
`SKPaymentTransaction-StatePurchased`	Transaction succeeded	Provide the purchased functionality. Validate the receipt, as described in *Receipt Validation Programming Guide*, and download any content, as described in "Providing Purchased Content" (page 27).
`SKPaymentTransaction-StateRestored`	Transaction ready to restore	For information about restoring transactions, see Restoring Transactions (page $@).

Listing 4-1 Responding to transaction statuses

```
- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray
*)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
            default:
                break;
        }
    }
}
```

To keep your user interface up to date while waiting, the transaction queue observer can implement optional methods from the `SKPaymentTransactionObserver` protocol as follows. The `paymentQueue:removedTransactions:` method is called when transactions are removed from the queue—in your implementation of this method, remove the corresponding items from your app's UI. The

`paymentQueue:restoreCompletedTransactionsFailedWithError:` or `paymentQueueRestoreCompletedTransactionsFinished:` method is called when Store Kit finishes restoring transactions, depending on whether there was an error. In your implementation of these methods, update your app's UI to reflect the success or error.

Persisting the Purchase

After making the product available, your app needs to make a persistent record of the purchase, to ensure that it continues to make the product available to the user in the future. Depending on the product's type and the versions of iOS your app supports, you may need to explicitly persist the purchases or the system may handle this for you.

- **Consumable products.** Your app updates its internal state to reflect the purchase, but there is no need to keep a persistent record because consumable products are not restored or synced across devices. Ensure that the updated state is part of an object that supports state preservation or that you manually preserve the state across app launches. For information about state preservation, see "State Preservation and Restoration" in *iOS App Programming Guide*.
- **Non-consumable products, auto-renewable subscriptions, and free subscriptions.** In iOS 7 and later, the system persists non-consumable products in the app receipt for you. If you need to support older versions of iOS, your app is responsible for keeping a persistent record of the purchase.
- **Non-renewing subscriptions.** Persisting the purchase is your app's responsibility on all versions of iOS and OS X. Store the receipt outside the device so that you can restore transactions across devices.

Information about consumable products and non-renewing subscriptions is added to the receipt when they are paid for, and remains in the receipt until you finish the transaction. After you finish the transaction, this information is removed the next time the receipt is updated—for example, the next time the user makes a purchase.

For information about the app's receipt, including how to read it and how to validate it, see *Receipt Validation Programming Guide*

Persisting Using User Defaults or iCloud

The User Defaults system is the simplest approach, but it doesn't sync across devices. In OS X, because users can edit the user defaults using Terminal, it may be more appropriate for your app to store a copy of the receipt in the user defaults—the receipt is signed and can't be tampered with, unlike simple Boolean or integer values.

To persist the purchase with the User Defaults system, set the value of a key.

```
NSUserDefaults * defaults = [NSUserDefaults standardUserDefaults];

// Store a simple value.
[defaults setBool:YES forKey:@"enable_rocket_car"];
[store setInteger:@15 forKey:@"highest_unlocked_level"];

// Update an array of receipts.
NSArray * receipts = [defaults arrayForKey@"receipts"];
NSData * newReceipt;  // Assume this exists.
NSArray * updatedReceipts = [receipts arrayByAddingObject:newReceipt];
[defaults setObject:newReceipts forKey@"receipts"];

[defaults synchronize];
```

To persist the purchase with iCloud, set the value of a key in the key-value store—the code in this approach is almost identical to the code for using the User Defaults system except you use an instance of `NSUbiquitousKeyValueStore` instead of `NSUserDefaults`.

```
NSUbiquitousKeyValueStore * store = [NSUbiquitousKeyValueStore defaultStore];

// Store a simple value.
[store setBool:YES forKey:@"enable_rocket_car"];
[store setInteger:@15 forKey:@"highest_unlocked_level"];

// Update an array of receipts.
NSArray * receipts = [store arrayForKey@"receipts"];
NSData * newReceipt;  // Assume this exists.
NSArray * updatedReceipts = [receipts arrayByAddingObject:newReceipt];
[store setArray:newReceipts forKey@"receipts"];

[store synchronize];
```

Persisting Using Your Own Server

To store the receipts on your server, send a copy of the receipt to your server along with some kind of credentials or identifier so that you can keep track of which receipts belong to a particular user. For example, let users identify themselves to your server with an email or user name, plus a password. Don't use the `identifierForVendor` property of `UIDevice`—you can't use it to identify and restore purchases made by the same user on a different device, because different devices have different values for this property.

Making the Product Available

Your app needs to take the appropriate actions to make the purchased product available to the user—unlock new app functionality, perform the purchased service, download now content, and so on. The exact details of how you make a product available depend on your app and the product.

If the product has associated content, your app needs to deliver that content to the user. For example, purchasing a level in a game requires delivering the files that define that level, and purchasing additional instruments in a recording app requires delivering the sound assets that the app needs to let the user play those instruments. For information about how to provide content, see "Providing Purchased Content" (page 27).

Finishing the Transaction

Finishing a transaction tells Store Kit that you have completed everything needed for the purchase. Unfinished transactions remain in the queue until they are finished, and the transaction queue observer is called every time your app is launched so that your app can finish them.

Complete all the following actions before you finish the transaction:

- Persist the purchase if needed.
- Download any purchased content.
- Update your app's UI to let the user access the product.
- Finish any other work needed to deliver the product.

To finish a transaction, call its `finishTransaction:` method. After you finish a transaction, do not take any actions on that transaction or do any work to deliver the product. If there is work remaining on a transaction, your app isn't ready to finish the transaction yet.

Note: Don't try to finish transactions immediately and use some other mechanism in your app to track them as actually unfinished. Store Kit is not designed to be used this way. Doing so would prevent your app from downloading hosted content, and could lead to other issues.

Suggested Testing Steps

Test each part of your code to verify that you have implemented it correctly.

Test a Successful Transaction

Sign in to the iTunes Store with a test user account, and make a purchase in your app. Set a breakpoint in your implementation of the transaction queue observer's `paymentQueue:updatedTransactions:` method, and inspect the transaction to verify that its status is `SKPaymentTransactionStatePurchased`.

Set a breakpoint at the point in your code that persists the purchase, and confirm that this code is called in response to a successful purchase. Inspect the User Defaults or iCloud Key-Value Store, and confirm that the correct information has been recorded.

Verify That Transactions Are Finished

Locate where your app calls the `finishTransaction:` method. Verify that all work related to the transaction has been completed before the method is called, and that the method is called for every successful purchase.

Providing Purchased Content

There are several approaches for delivering content: You can include content in your app bundle, download content from your own server, or download content from Apple's servers. There is a design tradeoff between the competing desires to make content available to the user as quickly as possible after it is purchased, and the desire to keep the initial download size of the app small. If you include too little in your app bundle, the user must wait for even small purchases to download. If you include too much in your app bundle, the initial download of the app takes much longer, the space is wasted for users who don't purchase the corresponding products, and, if the app becomes larger than 50 megabytes, users won't be able to download it over a cellular network.

> **Important:** You cannot patch your app binary or download executable code. Your app must contain all executable code needed to support all of its functionality when you submit it. If a new product requires code changes, you must submit an updated version of your app.

Embed small files (up to a few megabytes) in your app, especially if you expect most users to buy that product. This lets you make the purchased content available immediately. To add or update content in your app bundle, you have to submit an updated version of your app.

Host larger files on a server and download them when needed. You can add or update hosted content without needing to submit an updated version of your app (assuming your app uses a dynamic list of products). You can use hosted content associated with a free non-consumable purchase to deliver additional content while keeping your app's initial download small. For example, a game can include the first level it its app bundle and let users download the rest of the levels for free from Apple's servers.

When you host content on Apple's servers you don't need to worry about the server infrastructure. It is built on the same infrastructure that supports other large-scale operations, such as the iTunes Store, so your app can depend on it to be robust and scalable. Hosted content automatically downloads in the background even if your app isn't running.

When restoring transactions, let the user control what content gets downloaded, as described in Restoring Transactions (page $@).

Unlocking Local Content

To unlock local content after the user purchases it, load it using the `NSBundle` class, just as you load other resources from your app bundle.

```
NSURL * url = [[NSBundle mainBundle] URLForResource:@"rocket"
                                      withExtension:@"plist"];
[self loadCarAtURL:url];
```

Downloading Hosted Content from Apple's Server

> **Note:** For information about associating Apple-hosted content with a product, see "Creating Products in iTunes Connect" (page 8).

When the user purchases a product that has associated hosted content, the transaction passed to your transaction queue observer also includes an instance of `SKDownload` which lets you download the content.

To download the hosted content, add the download objects from the transaction's `downloads` property to the transaction queue by calling the `startDownloads:` method of `SKPaymentQueue`. If the value of the `downloads` property is `nil`, there is no hosted content for that transaction. Unlike downloading apps, downloading hosted content does not automatically require a Wifi connection for content larger than a certain size. Avoid using cellular networks to download large files without an explicit action from user.

Implement the `paymentQueue:updatedDownloads:` method on the transaction queue observer to respond to changes in a download's state—for example, by updating progress in your UI or by notifying the user of a failed download.

Update your user interface while the content is downloading using the values of the `progress` and `timeRemaining` properties. You can use the `pauseDownloads:`, `resumeDownloads:`, and `cancelDownloads:` methods of `SKPaymentQueue` from your UI to let the user control in-progress downloads. Use the `downloadState` property to determine whether the download has completed. Don't use the progress or time remaining, these properties wouldn't let your app accurately tell the difference between downloads that are almost finished and downloads that are completely finished.

After the download finishes, use its `contentURL` property to locate the content.

Note: Download all hosted content before finishing the transaction. After a transaction is complete, its download objects can no longer be used.

Downloading Content from Your Own Server

As with all other interactions between your app and your server, the details and mechanics of this process are up to you. The communication consists of, at minimum, the following steps:

1. Your app sends the receipt to your server and requests the content.
2. Your server validates the receipt to establish that the content has been purchased, as described in *Receipt Validation Programming Guide*.
3. Assuming the receipt is valid, your server responds to your app with the content.

Consider the security implications of how you host your content and of how your app communicates with your server. For more information, see *Security Overview*.

Locating and Managing Downloaded Content

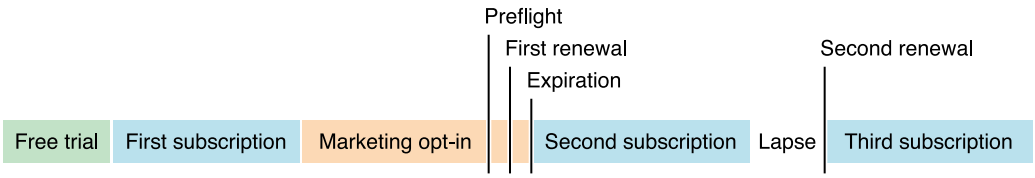
In iOS, your app can manage hosted content after the files are downloaded. The files are saved for you by the Store Kit framework in the `Caches` directory with the backup flag unset. In most cases, this is the correct behavior—because downloaded content can be re-downloaded, it can be deleted if the user becomes low on disk space (and later restored when disk space becomes available), and it should not be included in the user's backups. If your app's downloaded content should not be purgeable, move it to another location. If it should be included in the user's backup, set the backup flag.

In OS X, hosted content is saved and managed by the system; your app can't move or delete the files directly. To locate the file after downloading the content, use the `contentURL` property of the download object. To locate the file on subsequent launches, use the `contentURLForProductID:` class method of `SKDownload`. To delete a file, use the `deleteContentForProductID:` class method. For information about reading the product identifiers from your app's receipt, see *Receipt Validation Programming Guide*.

Working with Subscriptions

Apps that use subscriptions have some additional behaviors and considerations. Subscriptions incorporate an element of time, and your app needs to have the appropriate logic to determine whether the subscription is currently active and what time periods the subscription was active in the past. Your app also needs to react to new and renewed subscriptions, and properly handle expired subscriptions. Figure 6-1 (page 30) show an example subscription timeline, including some of the complexities your app needs to handle.

Figure 6-1 Example subscription timeline



Calculating a Subscription's Active Period

> **Note:** Do not calculate the subscription period by adding a subscription duration to the purchase date. This approach fails to take into account the free trial period, the marketing opt-in period, and the content made available immediately after the user purchased the subscription.

Your app needs to determine what content the user has access to based on the period of time that the subscription was active. Consider, for example, a magazine that publishes a new issue on the first day of each month. On February 7, the user buys a 3 month subscription When the user buys the subscription, the February issue is delivered immediately. In addition, the user gets access to all of the issues that are published between the subscription's start and end dates. Finally, on April 7, the subscription expires. There are a total of three issues available to the user—February, March, and April—as shown in Table 6-1.

Table 6-1 Timeline of a sample subscription

Magazine issue	User has access to content?
January 1	No, subscription has not started
February 1	Yes, delivered when the user subscribed

Magazine issue	User has access to content?
March 1	Yes, delivered at the time of its publication
April 1	Yes, delivered at the time of its publication
May 1	No, subscription expired on April 7

To implement this logic in your app, keep a record of the date that each piece of content is published. The user is has access to all content published between the start and end dates, as well as the content that was initially unlocked when the subscription was purchased.

Expiration and Renewal

The renewal process begins with a "preflight" check, starting ten days before the expiration date. During those ten days, the store checks for any issues that might delay or prevent the subscription from being automatically renewed—for example, if the customer no longer has an active payment method, if the product's price increased since the user bought the subscription, or if the product is no longer available. The store notifies users of any issue so that they can resolve it before the subscription needs to renew, ensuring their subscription is not interrupted.

During the 24 hour period before the subscription expires, the store starts trying to automatically renew it. The store makes several attempts to automatically renew the subscription, but eventually stops if there are too many failed attempts.

The store renews the subscription slightly it expires, to ensure that there there is no lapse in the subscription. However, lapses are still possible. For example, if the user's payment information was no longer valid, then the first renewal attempt would fail. If the user didn't update this information until after the subscription expired, there would be a short lapse in the subscription between the expiration date and the date that a subsequent automatic renewal succeeded. The user can also disable automatic renewal and intentionally let the subscription expire, then renew it at a later date, creating a longer lapse in the subscription. Make sure your app's subscription logic can handle lapses of various durations correctly.

> **Note:** Increasing the price of a subscription doesn't disable automatic renewal for all customers, only for those customers whose subscription expires in the next ten days. If this change is a mistake, restoring the original price means that no additional users are affected. If this change is intentional, keeping the new higher price will cause automatic renewal to be disabled for the rest of your users in turn as they enter the ten-day renewal window.

After a subscription is successfully renewed, Store Kit adds a transaction for the renewal to the transaction queue. Your app checks the transaction queue on launch, and handles the renewal the same way as any other transaction. Note that if your app is already running when the subscription renews the transaction observer is *not* called; your app finds out about the renewal the next time it is launched.

Cancellation

A subscription is paid for in full when it is purchased and can be refunded only by contacting Apple customer service. For example, if the user accidentally bought the wrong product, customer support can cancel the subscription and issue a refund. It is not possible for customers to change their mind in the middle of a subscription period and decide they don't want to pay for the rest of the subscription.

To check whether a purchase has been canceled, look for the cancellation date in the receipt. If the field is missing or has no date, the purchase is not canceled. If the field has a date in it, regardless of the subscription's expiration date, the purchase has been canceled—treat a canceled receipt the same as if no purchase had ever been made.

Depending on the type of product, you may be able to check only the currently active subscription, or you may need to check all past subscriptions. For example, a magazine app would need to check all past subscriptions to determine which issues the user had access to.

Cross-Platform Considerations

Product identifiers are tied to a specific platform. If you want to let users who have a subscription in an iOS app access the content from an OS X app (or vice versa), that is up to your apps. You will need to have some system for identifying users and keeping track of what content they have subscribed to, similar to what you would implement for an app that uses non-renewable subscriptions.

The Test Environment

For the sake of testing, there are some differences in behavior between auto-renewable subscriptions in the production environment and in the test environment.

Renewal happens at an accelerated rate, and auto-renewable subscriptions renew a maximum of six times per day. This lets you test how your app handles subscription renewal and how it handles a renewal after a lapse and how it handles a subscription history that includes gaps.

Because of the accelerated expiration and renewal rate, the subscription can expire before the system starts trying to renew the subscription, leaving a small lapse in the subscription period. Such lapses are also possible in production for a variety of reasons—make sure your app handles them correctly.

Preparing for App Review

After you finish testing, you are ready to submit your app for review. This chapter highlights a few tips to help you through the review process.

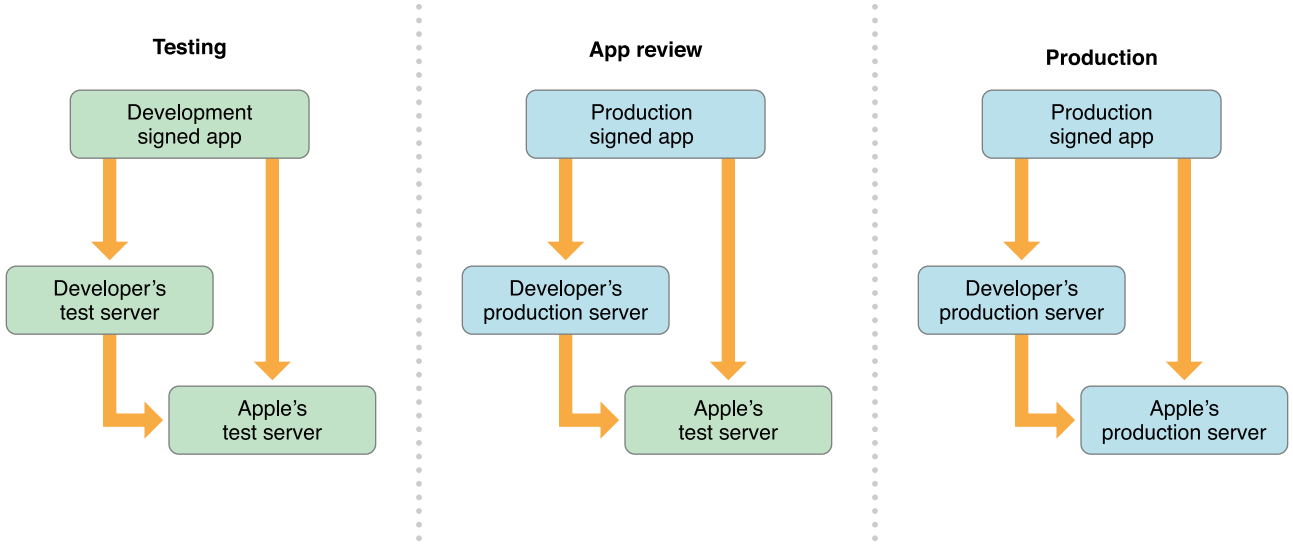
Submitting Products for Review

The first time you submit your app for review, you also need to submit in-app products to be reviewed at the same time as your app. After the first submission, you can submit updates to your app and products for review independently of each other.

Receipts in the Test Environment

Your app runs different environments while in development, review, and production, as show in Figure 7-1 (page 34).

Figure 7-1 Development, review, and production environments



During development, you run a development signed version of your app, which connects to your development servers and the test environment for the App Store. In production, your users run a production signed version of your app which connects to your production servers and the real store. However, during app review, your app runs in a mixed production/test environment: it is production signed and connects to your production servers, but it connects to the test environment for App Store.

When validating receipts on your server, your server needs to be able to handle a production-signed app getting its receipts from Apple's test environment. The recommended approach is for your production server to always validate receipts against the production store environment first. If validation fails with an error code 20117 "Sandbox receipt used in production", validate against the test environment instead.

Document Revision History

This table describes the changes to *In-App Purchase Programming Guide*.

Date	Notes
2013-09-18	Expanded and reorganized content throughout. Moved information about validating receipts to *Receipt Validation Programming Guide*.
2012-09-19	Removed note that expires_date key was not present on restored transactions. Best practice for restoring auto-renewable subscriptions is to simply respect the expires_date key on the restored transactions. Removed section on restoring auto-renewable subscriptions that indicated otherwise.
2012-02-16	Updated artwork throughout to reflect cross-platform availability. Updated code listing to remove deprecated method. Replaced the deprecated `paymentWithProductIdentifier:` method with the `paymentWithProduct:` method in "Adding a Store to Your Application" (page $@).
2012-01-09	Minor updates for using this technology on OS X.
2011-10-12	Added information about a new type of purchase to the overview.
2011-06-06	First release of this document for OS X.
2011-05-26	Updated to reflect the latest server behavior for auto-renewable subscriptions.
2011-03-08	Corrected the list of keys in the renewable subscriptions chapter.

Date	Notes
2011-02-15	Apps must always retrieve product information before submitting a purchase; this ensures that the item has been marked for sale. Information added about renewable subscriptions.
2010-09-01	Minor edits.
2010-06-14	Minor clarifications to SKRequest.
2010-01-20	Fixed a typo in the JSON receipt object.
2009-11-12	Receipt data must be base64 encoded before being passed to the validation server. Other minor updates.
2009-09-09	Revised introductory chapter. Clarified usage of receipt data. Recommended the iTunes Connect Developer Guide as the primary source of information about creating product metadata. Renamed from "Store Kit Programming Guide" to "In App Purchase Programming Guide".
2009-06-12	Revised to include discussion on retrieving price information from the Apple App Store as well as validating receipts with the store.
2009-03-13	New document that describes how to use the StoreKit API to implement a store with your application.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010